The Report Committee for Juan Trejo Martinez
Certifies that this is the approved version of the following Report

Battery Vent Gas Hazard Analysis and Building Deflagration Dashboard Development

APPROVED BY
SUPERVISING COMMITTEE:

Ofodike Ezekoye, Supervisor

Matthew Hall

Battery Vent Gas Hazard Analysis and Building Deflagration Dashboard Development

by

Juan Trejo Martinez

Report

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Master of Science in Engineering

The University of Texas at Austin

December 2019

Abstract

Battery Vent Gas Hazard Analysis and Building Deflagration Dashboard Development

Juan Trejo Martinez, M.S.E.

The University of Texas at Austin, 2019

Supervisor: Ofodike A. Ezekoye

Lithium-ion battery failures can lead the battery cell to undergo what's known as thermal runaway, resulting in a potentially catastrophic fire or explosion. During a thermal runaway event a series of chemical reactions take place, increasing the cell temperature and resulting in the generation of a flammable gas mixture.

The University of Texas Fire Research Group has developed two models to evaluate fire and explosion hazard for lithium-ion batteries using the aforementioned studies. The first model aims to estimate the upper and lower flammability limits, laminar flame speed, and maximum overpressure of the gases released during thermal runaway. While the second model predicts the pressure time history of an explosion. Both of these models are intended to provide a framework for evaluating the overall safety of lithium-ion cells.

To enable our researches to perform rapid data analysis as more experimental data becomes available, a web application that presents the above-mentioned models in the form

of interactive dashboards was developed and deployed using Python and a framework for building data visualization web applications called Dash.

The first part of development was building the application's layout. The layout of a Dash application is built by assembling a hierarchical tree of components. These components are the building blocks of the application and can come from either the *dash_html_components* or *dash_core_components* modules. The second part of development involved creating and wiring up a MongoDB database to the application. This connection allows the application to access and perform operations on the stored data.

Finally, deploying the application was done through Heroku, a Platform as a Service that provides the infrastructure and user interface to publish and manage a web application.

The two dashboards deployed are the Vent Gas Hazard Analysis dashboard and the Building Deflagration dashboard. The Vent Gas Hazard Analysis dashboard summarizes key gas characteristics such as lower and upper flammability limits, laminar flame speed, and maximum over-pressure, which are essential in quantifying the overall hazard potential or the runaway event. While the Building Deflagration dashboard provides information on how room pressure changes over a period of time during a battery explosion event.

Table of Contents

List of Figures

Chapter 1: Introduction

Lithium-ion battery failures, stemming from manufacturing defects, thermal or electrical abuse, and/or mechanical damage, can lead the cell to undergo what's known as thermal runaway, resulting in potentially catastrophic fire or explosion scenarios that pose a significant risk to surrounding life and property. During a thermal runaway event a series of chemical reactions take place, increasing the cell temperature and resulting in the generation of a flammable gas mixture, which can build up and rupture the cell.

Through experimental testing, multiple studies have characterized the gas composition, flammability limits, and maximum pressure for battery cells of differing constructions and chemistries at varying states-of-charge (SOC) while undergoing thermal runaway.

The University of Texas Fire Research Group has developed two models to evaluate fire and explosion hazard for lithium-ion batteries using the aforementioned studies. The first model aims to estimate the flammability limits, laminar flame speed, and maximum overpressure of the gases released during thermal runaway. While the second model predicts the pressure time history of an explosion. Both of these models are intended to provide a framework for evaluating the overall safety of lithium-ion cells.

As more experimental data becomes available, the researches at the Fire Research Group need a tool to help them visualize and analyze the incoming data to reach new insights regarding the safety of lithium-ion cells. In order the enable our researches to perform this rapid data exploration, a web application that presents above-mentioned models in the form of interactive dashboards was developed. This body of work describes the process of building and deploying this application.

Chapter 2: Web Application Development

2.1 TECHNOLOGIES

An array of technologies was used during the development of the web application. A brief description of the primary ones is presented in this section.

2.1.1 Python

Python is high-level, general-purpose, object-oriented programming language. While it is used in a wide variety of fields, given its elegant syntax and dynamic typing, Python has become particularly popular within the scientific community due to its high-performance nature. Python offers a plethora on third-party scientific computing packages, typically written in low-level languages like C, or Fortran for their high performance. Some of the most popular libraries used are NumPy, SciPy and Matplotlib, to name a few. The aforementioned reasons make Python a clear language choice in the development of the web application.

2.1.2 Dash

Dash is a framework for building data visualization web applications using Python. Dash abstracts technologies such as JavaScript, React.js, and Flask to streamline the process of building an interactive web application. In addition, Dash is compatible with Plotly, which is an open-source graphing Python library that renders interactive, publication-quality graphs.

Because Dash apps are web-based applications, they can be easily deployed to a personal server or through cloud platform services such as Heroku, Amazon Web Services (AWS), or Google Cloud Platform (GCP). Once deployed, the apps are then available via the public internet and can be accessed from any location using a web browser.

Dash was the ideal choice for building a data visualization tool given its features and its highly custom user interfaces written in pure Python.

2.1.3 Mongo DB

Unlike standard relational databases, such as SQLite, MySQL, or Postgres, where data is stored in multiple tables composed of rows and columns, MongoDB is a document database, where data is stored in JSON-like documents. This document model allows the data structure and data fields to vary from document to document and even change over time, making the database both flexible and scalable. MongoDB also offers a Python API called PyMongo, which makes it possible interact with a remote database inside a Python application.

2.1.4 Conda

Conda is an open source, multi-platform Python environment manager used to install, run and update Python packages and their dependencies. With Conda, one can create, save, and load a Python environment on a local computer or a remote server. This is a key function during the development and testing of a Python application in order to have repeatable and reproduceable outcomes no matter the system or platform that is used.

2.2 BATTERY VENT GAS CHARACTERIZATION DATA

The source of data for the web application comes from multiple studies that have documented the type of gases released during thermal runaway of a battery. These studies, summarized in Appendix A0, generally consist in failing lithium-ion cells in either an enclosed chamber, or under an exhaust hood. The gases expelled by the failed battery are then extracted and analyzed. Some of the measurements made during these tests include vent gas composition, flammability limits and maximum pressure for battery cells of differing constructions and chemistries at varying states-of-charge (SOC).

Our main dataset is the reported vent gas composition, shown in Figure 1, from which all our plots in our web application will be generated using various physical models.

Literature	Chemistry	SOC (%)
Ohsaki (2005)	LCO	OCH
Somandepalli (2014)	LCO	150
Somandepalli (2014)	LCO	100
FAA Study (2016)	LCO	100
FAA Study (2016)	LCO	90
FAA Study (2016)	LCO	80
FAA Study (2016)	LCO	70
FAA Study (2016)	LCO	60
Somandepalli (2014)	LCO	50
FAA Study (2016)	LCO	50
FAA Study (2016)	LCO	40
FAA Study (2016)	LCO	30
FAA Study (2016)	LCO	20
FAA Study (2016)	LCO	10
Kumai (1999)	LCO	0
Golubkov (2014)	NMC	100
Golubkov (2014)	LCO/NMC	100
Fernandes (2018)	LFP	190
Yuan (2015)	LFP	190
Yuan (2015)	LFP	180
Yuan (2015)	LFP	170
Golubkov (2015)	LFP	130
Golubkov (2015)	LFP	115
Golubkov (2015)	LFP	100
Golubkov (2014)	LFP	100
Golubkov (2015)	LFP	75
Golubkov (2015)	LFP	50
Golubkov (2015)	LFP	25
Golubkov (2015)	LFP	0
Zheng (2016)	LFP	0
Golubkov (2015)	NCA	143
Golubkov (2015)	NCA	132
Golubkov (2015)	NCA	127
Golubkov (2015)	NCA	120
Golubkov (2015)	NCA	112
Golubkov (2015)	NCA	104.6
Golubkov (2015)	NCA	100
Golubkov (2015)	NCA	100
Golubkov (2015)	NCA	100
Lammer (2017)	NCA-32A	100
Lammer (2017)	NCA-35E	100
Lammer (2017)	NCA-MJ1	100
Golubkov (2015)	NCA	75
Golubkov (2015)	NCA	50
Golubkov (2015)	NCA	0
Golubkov (2015)	NCA	0
Golubkov (2015)	NCA	0
Golubkov (2015)	NCA	0
Golubkov (2015)	NCA	0

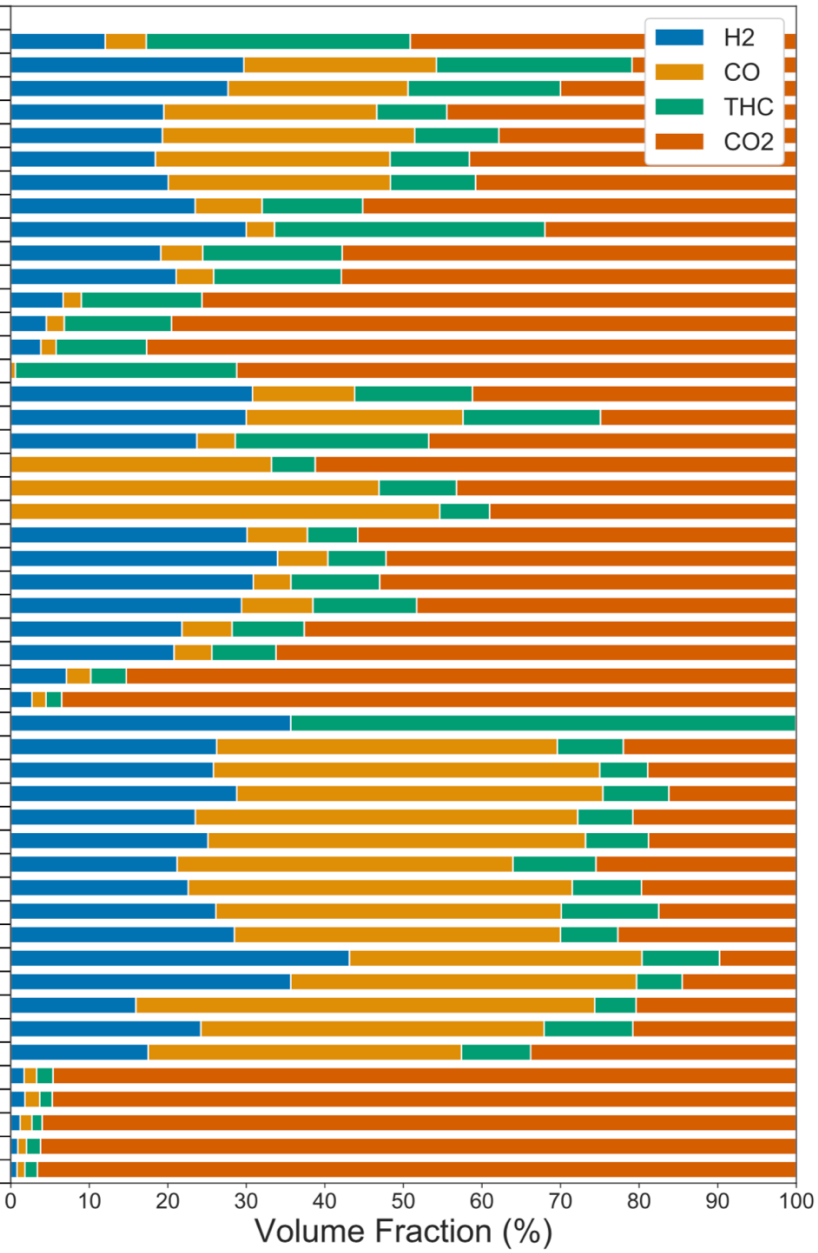


Figure 2.1: Battery gas species compositions by publication.

2.3 DEVELOPING THE WEB APPLICATION

The development of a web application can be distilled into two components: frontend development, which involves defining the appearance and flow of the user-interface (UI) of the application; and backend development which focus is around the flow of data in the application – how it's handled, where it's stored and what computations are be performed on it.

2.3.1 Frontend development

While the standard frontend development deals with technologies such as HTML, JavaScript, React.js, Bootstrap, etc. Dash abstracts away most of those technologies and provides a simple, yet powerful, API to create a data visualization dashboard.

2.3.1.1 Building a UI with Dash

When building a Dash application, the two elements to consider are: the "layout" of the application, which describes what the application looks like to the user; and the application's "interactivity", which determines how it reacts to the user's input.

Similar to when building an HTML application, the layout of a Dash application is built by assembling a hierarchical tree of components. These components are the building blocks of the application and can come from either the *dash_html_components* module (abbreviated as *html*), which provides classes for all of the HTML tags where the keyword arguments describe HTML attributes like *style*, *className*, and *id*; or from the *dash_core_components* module (abbreviated as *dcc*), which generates higher-level components such as controls and graphs.

To illustrate the above-mentioned concepts, a minimal working example of a Dash application is presented below.

```
# -*- coding: utf-8 -*-
import dash
import dash_core_components as dcc
import dash_html_components as html

external_stylesheets = ['https://codepen.io/chriddyp/pen/bWLwgP.css']

app = dash.Dash(__name__, external_stylesheets=external_stylesheets)

app.layout = html.Div(children=[
    html.H1(children='Hello Dash'),

    html.Div(children='''
        Dash: A web application framework for Python.
    '''),

    dcc.Graph(
        id='example-graph',
        figure={
            'data': [
                {'x': [1, 2, 3], 'y': [4, 1, 2], 'type': 'bar', 'name':
'SF'},
                {'x': [1, 2, 3], 'y': [2, 4, 5], 'type': 'bar', 'name':
u'Montréal'},
            ],
            'layout': {
                'title': 'Dash Data Visualization'
            }
        }
    )
])

if __name__ == '__main__':
    app.run_server(debug=True)
```

Figure 2.2: Python code of example Dash application.

The above code imports the required modules (*dcc* and *html*), creates an application object called *app*, and defines the application layout. This layout consists of a *html.Div* component, which defines the main section in the application, with three sub-components. These sub-components are a header (*html.H1*), a paragraph (*html.Div*) and a graph (*dcc.Graph*). Note that each of these components in turn define their state, such as the text in the header, or the data and layout of the graph.

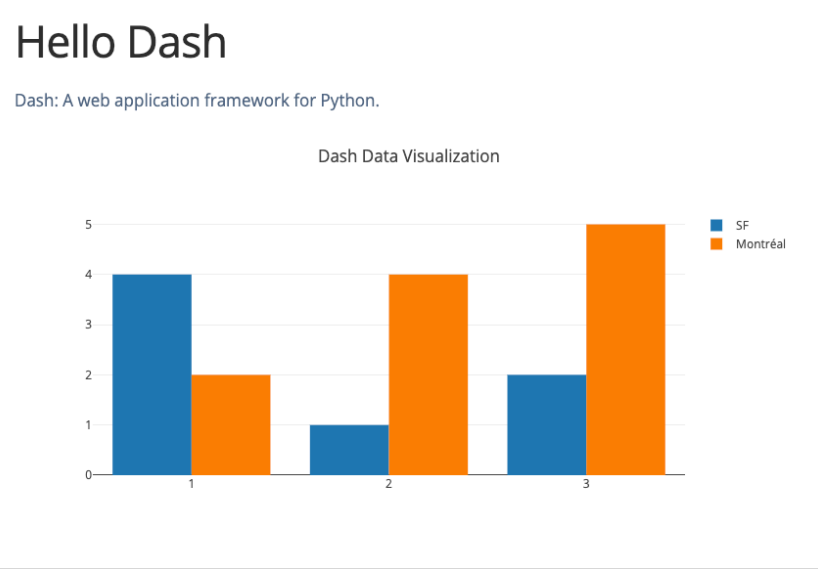


Figure 2.3: Browser view of example Dash application.

While the example is minimal, it includes the core elements required to build the layout of dashboard app and can easily be expanded by adding more components to the layout.

2.3.1.2 Dashboard Layout

Applying the concepts described in the previous section, building the layout of our dashboard was simply a matter of bringing Dash components together.

Our overall layout of the dashboard is depicted in Figure 2.4. As shown, there are three main sections in our dashboard: a header, a data input and selection section, and a data visualization section.

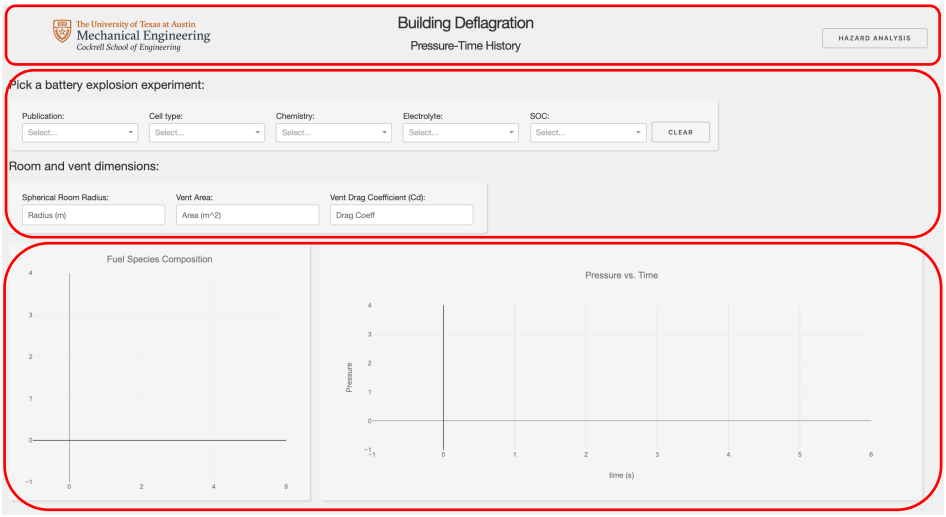


Figure 2.4: Application layout with three components: Header, Data Input and Data Visualization

The header section, shown in more detail in Figure 2.5, is composed of an image element (*html.Img*), two headers to indicate the title and subtitle of the dashboard (*html.H3*, *html.H5*) and a button (*html.Button*) to navigate between dashboards. The code used to generate this header can be found in Appendix A1.

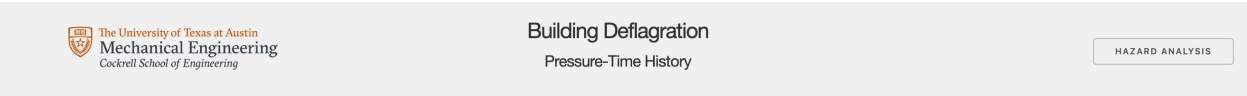


Figure 2.5: Header section.

The data input and selection section, shown in Figure 2.6, consists of mainly of dropdown menus (*dcc.Dropdown*), and input text boxes (*dcc.Input*) that allow the user to select and input data, respectively. Each of those input components is accompanied by paragraphs (*html.P*) and labels (*html.Label*) that inform the user about the purpose of the component. Finally, a *html.Button* is included to help the user clear all the dropdown selections. To view the code associated with this section refer to Appendix A2 and A3.

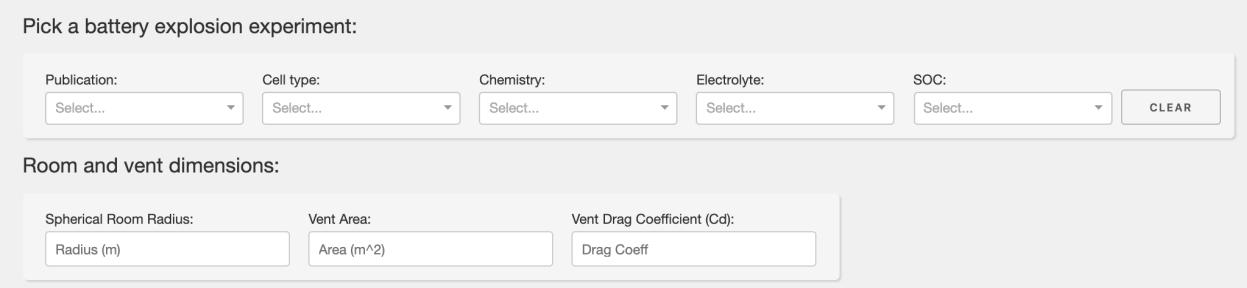


Figure 2.6: Data input and selection section.

Lastly, Figure 2.7 shows the data visualization section, which is entirely composed of *dcc.Graph* elements. As it can be seen from the code found in Appendix A4, these *dcc.Graph* component definitions act solely as placeholders for the plot. The data and layout of the plot (axes labels, plot title, etc.) are defined via Dash *callbacks*, which will be discussed in a future section.

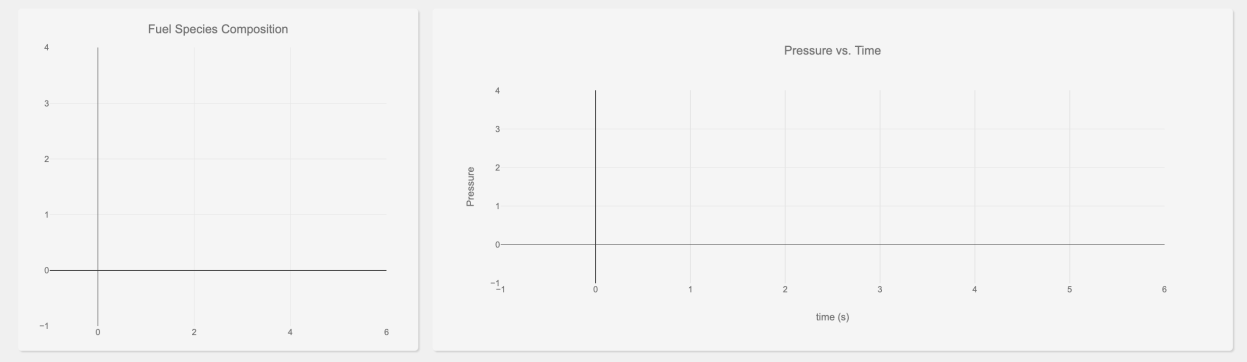


Figure 2.7: Data visualization section.

2.4.1 Backend development

Once the layout is defined, a web application needs to be interactive and react real-time to user input. This is done by wiring up the application to a database and coding up instructions on what operations to perform on the data.

2.4.1.1 Data Access and Storage

A remote MongoDB database was chosen to store the experimental data. Because this database lives on a server in the cloud, and not on a local machine, a connection needs to be created between the application and the database in order to access and interact with the data. This is seamlessly done through MongoDB's Python API called PyMongo. With PyMongo, one can interact with the remote database within the context of a Python application. This interaction can be summarized by what are referred to as Create, Read, Update and Delete, or CRUD, operations. Specifically, what PyMongo then enables the application to do is: add new data to the database ('create'), search for existing data in the database ('read'), modify existing data ('update'), and 'delete' any existing data entries.

Creating a connection between the application and the database requires only a few lines of code, as the example in Figure 2.8 shows:

```
from pymongo import MongoClient

# define database url
url = 'mongodb://mongodb0.example.com:27017/admin'

# connect to MongoDB client
client = MongoClient(url)

# access the `admin` database
db = client.admin
```

Figure 2.8: Python code connecting to a Mongo DB database.

Once connected to the database, one can access any data collections within the database and search and retrieve documents inside those collections via the *db* object. Because these CRUD operations are often required in multiple places across the application, it is a good idea to write a set of helper functions that abstract that behavior. Figure 2.9 shows an example of such function,

which queries the database for all the unique values for a given field in a collection. The remainder helper functions can be found in Appendix A5.

```
def get_unique(collection, field, search={}):
    """ Get all unique values for a given field in a collection.

    Parameters
    ----------
    collection : Unicode
        Name of te collection to search in.
    field : Unicode
        Name of the field to retrieve.
    search : Dict
        The search dictionary with the search parameters.

    Returns
    -------
    List
        List of unique values for a given field in a collection.
    """
    client = MongoClient(DB_URI)
    db = client[DB_NAME]

    return db[collection].find(search).distinct(field)
```

Figure 2.9: Example CRUD helper function.

2.4.1.2 Callbacks

Callbacks are what make a Dash application interactive. They enable application components, such as dropdowns, textboxes, and graphs, to "react" to user input by updating their contents or changing their appearance. This makes for a dynamic application with a more immersive user experience.

Callbacks work via the *@app.callback* decorator[1]. The decorator takes a list of *Input* objects and performs an operation, defined by the decorated function, on a list of *Output* objects.

[1]A decorator is a bit of Python syntactic sugar that allows us to wrap another function in order to extend the behavior of wrapped function, without permanently modifying it.

The code in Figure 2.10 illustrates how a callback was used to update the contents of the dropdown menus in the application.

```
@app.callback([Outputs], [Inputs])
def update_dropdowns(data, publication, cell_type, chemistry, electrolyte,
                     soc):
    """ Update gas dropdown databa based on selections. """
    df = pd.DataFrame(json.loads(data))

    if publication:
        df = df[df['Publication'] == publication]
    if cell_type:
        df = df[df['Format'] == cell_type]
    if chemistry:
        df = df[df['Chemistry'] == chemistry]
    if electrolyte:
        df = df[df['Electrolyte'] == electrolyte]
    if soc:
        df = df[df['SOC'] == soc]

    fields = ['Publication', 'Format', 'Chemistry', 'Electrolyte', 'SOC']
    results = []
    for field in fields:
        result = list(df[field].unique())
        results.append(make_options(result))
    return results
```

Figure 2.10: Example callback code to update the dropdown menus.

Where the *Inputs* are the selected dropdown values and the *Outputs* are the lists of available options for each dropdown.

The same pattern is followed for populating the *Graph* components as well as defining their appearance. This is shown in Figure 2.11 below.

```
@app.callback(Output("composition_plot", "figure"),
              [Input("gas_composition", "children")])
def make_gas_composition_plot(gases):
    """ Create gas composition plot. """
    gases = json.loads(gases) if gases else {}
    data = []

    if gases:
        fuel_species = _get_fuel_species(gases)

        data = [
            dict(
                type="pie",
                labels=[key for key, val in fuel_species.items() if val > 0],
                values=[val for val in fuel_species.values() if val > 0],
                name="Fuel Species Composition",
                textinfo="label",
                textfont=dict(size="18", color="#FFFFFF"),
                hoverinfo="label+percent",
                marker=dict(colors=[GAS_COLORS[gas] for gas in
                                    fuel_species]),
            ),
        ]

    layout = copy.deepcopy(plot_layout)
    layout["title"] = "Fuel Species Composition"
    layout["margin"] = dict(l=30, r=30, b=20, t=40)  # noqa
    layout["legend"] = dict(font=dict(color="#777777", size="12"),
                            orientation="h")

    figure = dict(data=data, layout=layout)
    return figure
```

Figure 2.11: Example callback to generate the gas composition pie chart.

The above code snippet shows the callback used to update the fuel species composition pie chart plot shown in Figure 11. The remainder of the callbacks used through the application can be found in Appendix A6. A figure of the pie chart can be found in a later section.

Chapter 3: Deployment

3.1 TECHNOLOGIES

This section briefly introduces the tools and technologies used in the deployment of the web application.

3.1.1 Heroku

Heroku is a cloud computing platform, also known as platform as a service (PaaS), for deploying and running web applications. PaaS providers, such as Heroku, AWS, or GCP, provide an infrastructure composed of both hardware and software that allow customers to develop, publish and manage their applications. Leveraging this infrastructure minimizes the cost and complexity associated with launching a web application.

3.1.2 Docker

Docker is a containerization service that 'packages' software into 'containers' for development and deployment. This container is a standard unit of software that packages up the code in an application so that it runs quickly and reliably regardless of the computing environment.

3.1.3 Travis CI

Travis CI is a continuous integration service which enables developers to automatically and continuously test and build software on multiple platforms throughout the development process. This ensures that the code is in a good state prior to deployment.

3.1.4 Git & GitHub

Git is the ubiquitous version control tool to manage source code history by tracking changes in any set of files, enabling software teams to coordinate work among programmers. GitHub is a Git repository hosting service, meaning code lives on a remote location that can be accessed by anyone (if made public) or by a team (if kept private). While Git is a command line tool, GitHub provides a Web-based graphical interface.

3.2 DEPLOYING THE WEB APPLICATION

In order to deploy a web application, there are some technical considerations that need to be kept in mind. What infrastructure, i.e. the hardware and software, will be used to host and run the application? How will the data storage, networking, performance monitoring, and security of the application be handled? These can be hard questions to answer for someone deploying their first application. PaaS exist to handle all the technical complexity and enable developers to seamlessly deploy and manage an application. Out of all the available PaaS, Heroku is one of the most user-friendly services, making it a great choice for first time deployments.

3.2.1 Deploying with Heroku

Deploying an application with Heroku can be completed in a few steps and within minutes of creating a free Heroku account at signup.heroku.com.

The first step to deploy with Heroku is to download and install the Heroku Command Line Interface (CLI). The CLI is used to deploy, manage and scale the application, provision add-ons, view application logs, and run the application locally. The CLI can be downloaded at devcenter.heroku.com/articles/heroku-cli. Once installed, the *heroku* command can be used from the command shell to log in to the Heroku CLI.

```
$ heroku login
heroku: Press any key to open up the browser to login or q to exit
 ›   Warning: If browser does not open, visit
 ›   https://cli-auth.heroku.com/auth/browser/***
heroku: Waiting for login...
Logging in... done
Logged in as me@example.com
```

Figure 3.1: Heroku CLI login command.

Next, the application needs to be prepared for deployment by defining a *Procfile*[2] and a *requirements.txt*[3] file in the working directory. These files will tell Heroku how to run the application and what Python packages the application depends on, respectively. With those two files defined, a local copy of the source code must be made available for deployment.

```
$ git clone https://github.com/my_repo/my-web-app.git
$ cd my-web-app
```

Figure 3.2: Cloning source code from GitHub.

Now the application is ready for deployment. To do so, one must first create an app on Heroku, which prepares Heroku to receive the source code.

```
$ heroku create my-first-app
Creating app... done, ⬢ my-first-app
https:// my-first-app.herokuapp.com/ | https://git.heroku.com/ my-first-app.git
```

Figure 3.3: Creating an app with Heroku.

The snippet above creates an app called *my-first-app*, when doing so, Heroku associates a git remote called *heroku* with the local repository. This allows Heroku to track any future changes made to the application.

[2] A Procfile specifies the commands that are executed by the app on startup. It can be used to declare a variety of process types, including: the app's web server, multiple types of worker processes, a singleton process, such as a clock and a list of asks to run before a new release is deployed.

[3] The requirements.txt file lists all of the application's package dependencies. When an app is deployed, Heroku reads this file and installs the appropriate Python dependencies using the *pip install -r* command.

Finally, to publish the application one can simply run the following command:

```
$ git push heroku master
. . .
remote: Verifying deploy... done.
To https://git.heroku.com/serene-caverns-82714.git
 * [new branch]      revert-to-requirements -> master
```

Figure 3.4: Deploying the Heroku app.

Now to visit the application at the URL generated by Heroku using the application's name the website can be opened with the following command within the application's directory:

```
$ heroku open
```

Figure 3.5: Opening the deployed Heroku app on a browser.

3.2.2 The Need for Docker

While Heroku is a powerful and easy to use service, it has its limitations. Primarily, it does not give the user control over the operating system (OS) and the Python packages that can be installed in it. Heroku installs Python dependencies via *pip*, which is the de facto package-management system used to install Python packages. This, however, limits the ability to install packages **not** provided by *pip*. So rather than being constrained by Heroku's environment, one can have full control over the OS and install any Python package by using Docker.

Deploying an application to Heroku with Docker, while still simple, it differs from the steps described in section 3.2.1. Mainly, in order to deploy an application with Docker, the

application source code needs to include a *Dockerfile*. This *Dockerfile* is a text document that contains all the commands a user could call on the command line to assemble a Docker image[4]. The Docker file used in our web application can be found in Appendix A7.

With the Docker file in place, one can then deploy to Heroku via the Heroku CLI, just as before. To run the commands below one needs to have a working Docker installation and be logged in to Heroku (*heroku login*).

The first step in the deployment process is to log in to the Container Registry. The Container Registry is a repository where Docker images are stored:

```
$ heroku container:login
```

Figure 3.6: Logging into the Container Registry.

Then, one must build the image locally, and push it to the Container Registry via:

```
$ heroku container:push web
```

Figure 3.7: Building and pushing the Docker image to the Container Registry.

Finally, the app is deployed by releasing the image to the application:

```
$ heroku container:release web
```

Figure 3.8: Deploying the web application.

[4] A Docker image is made up of multiple layers. A user composes each Docker image to include system libraries, tools, and other files and dependencies for the executable code.

Just as before, the application can be viewed in a browser with:

```
$ heroku open
```

Figure 3.9: Opening the deployed Heroku app on a browser.

Chapter 4: Web Application

4.1 DASHBOARDS

Our web application consists primarily of two dashboards. While both dashboards share some similarities, they each serve a different purpose. This section briefly describes each dashboard, and their respective components.

4.1.1 Building Deflagration Dashboard

The purpose of the Building Deflagration dashboard is to depict how room pressure changes over a period of time during a battery explosion event. This pressure is a function of the type of battery (cell type, chemistry, electrolyte, and state of charge), the dimensions of the enclosure where the explosion takes place, and the vent size and specifications. A full picture of the dashboard is show in Appendix A8

4.1.1.1 Data Input

As mentioned above, there are multiple inputs required to generate a room pressure vs time plot. The first input is the battery specifications, which can be selected via the dropdown menus, shown in Figure 21.

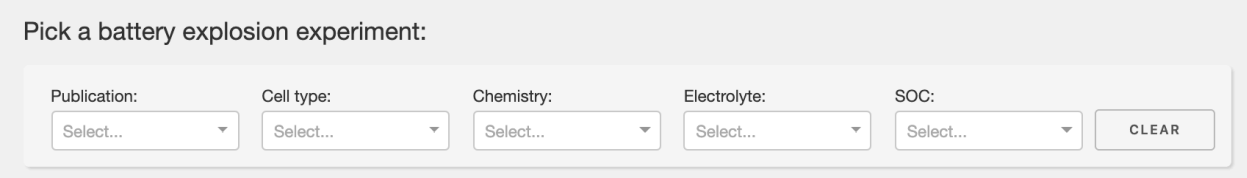


Figure 4.1: Dropdown menus for selecting a battery type.

The dropdowns allow the user to select by a battery by: the publication where the battery gases released during thermal runaway were first characterized through experimental testing, the battery cell type (pouch cells, 18650, cylindrical), the cell chemistry (Lithium Cobalt Oxide, Lithium Iron Phosphate, etc.), the cell electrolyte, and the battery's state of charge (%). While each

of the dropdowns display all available options at first, as selections are made the adjacent dropdowns filter their options to the ones corresponding to the selections.

Room dimensions, vent size and vent drag coefficient can be entered via the input text boxes shown below. Each box will validate the input to ensure that reasonable values are entered by the user.

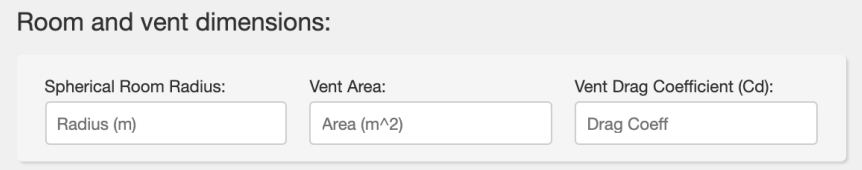


Figure 4.2: Input boxes to define room and vent specifications.

4.1.1.2 Fuel Species Composition Plot

The Fuel Species Composition plot is there to show the user the composition of the flammable gasses expelled by the battery during thermal runaway. An example of the plot is shown below.

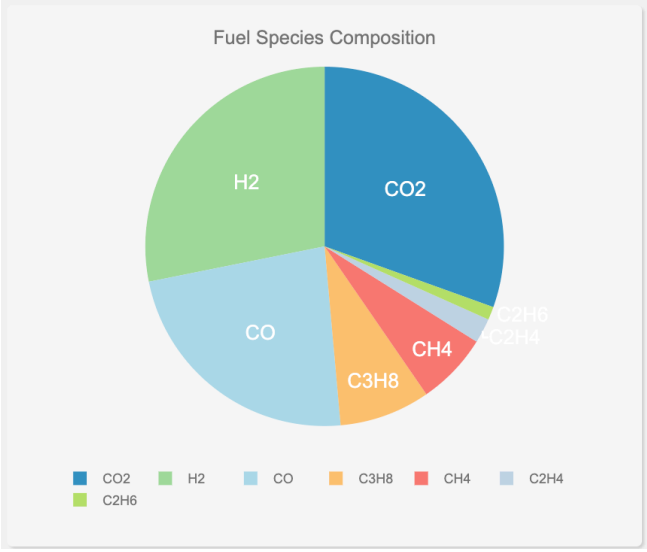


Figure 4.3: Fuel Species Composition plot.

4.1.1.3 Pressure-Time History Plot

The plot below shows the pressure-time history an indoor explosion inside a vented enclosure. This plot is generated by a vented enclosure explosion model that relies on both properties of the gas mixture, and the geometry and venting of the enclosure, to calculate the pressure time history produced by deflagrations of flammable gas mixtures.

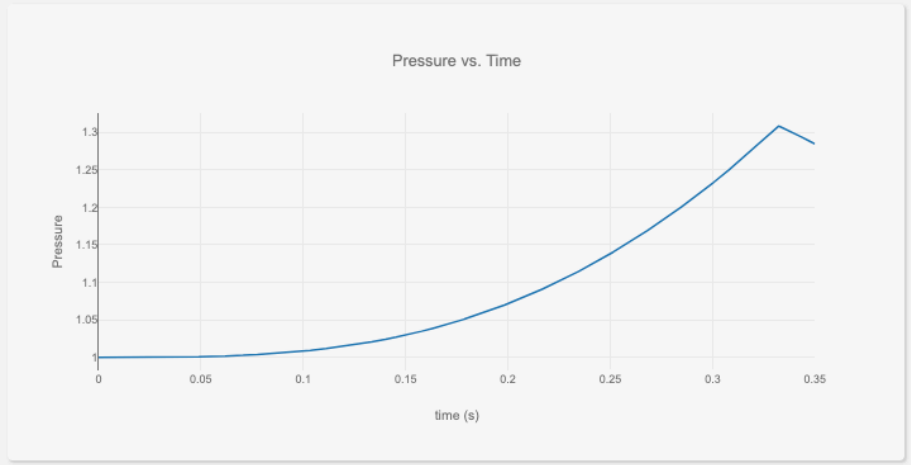


Figure 4.4: Pressure-time history plot.

4.1.2 Battery Vent Gas Hazard Analysis Dashboard

The Vent Gas Hazard Analysis dashboard provides insights on the flammability properties of the gases released during a thermal runaway of a battery cell. Specifically, the dashboard summarizes key gas characteristics such as lower and upper flammability limits, laminar flame speed, and maximum over-pressure, which are essential in quantifying the overall hazard potential or the runaway event.

4.1.2.1 Summary Table

Figure 4.5 below summarizes the lower and upper flammability limits (LFL & UFL), maximum laminar flame speed, and maximum overpressure. The LFL and UFL are important in a hazard analysis because they define the volume fraction of fuel required to create the potential for a fire or explosion. The flame speed is also a relevant parameter as it determines whether the

flammable mixture burns in a non-premixed or premixed mode, where the explosion hazard scenario occurs when the vented gases burn in a premixed mode. Finally, the maximum adiabatic pressure is the pressure that is generated when the gas is combusted in a perfectly adiabatic, constant volume process. This pressure defines the maximum possible pressure that the gas could possibly generate.

Parameter	Value
Lower Flammability Limit	7.00 %
Upper Flammability Limit	38.00 %
Laminar Flame Speed	0.49 m/s
Max Adiabatic Pressure	8.37 bar

Figure 4.5: Summary table of LFL, UFL, flame speed and max pressure

4.1.2.2 Ternary Contour Plot

The ternary plot is also included to show how different parameters chance as a function of fuel, air and inert concentrations. In Figure 4.6 below, the ternary plot for equivalence ratio, which is the mass ratio of fuel to air compared to the stoichiometric mass ratio of fuel to air, is shown.

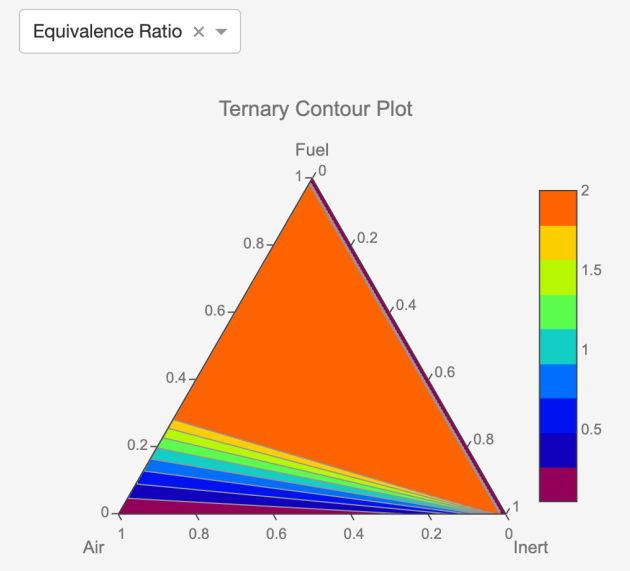


Figure 4.6: Equivalence ratio ternary contour plot

Figure 4.7 shows the adiabatic flame temperature, which is the is the temperature that results from a complete combustion process that occurs without any work, heat transfer or changes in kinetic or potential energy. Finally, Figure 4.8 shows the flammability limits.

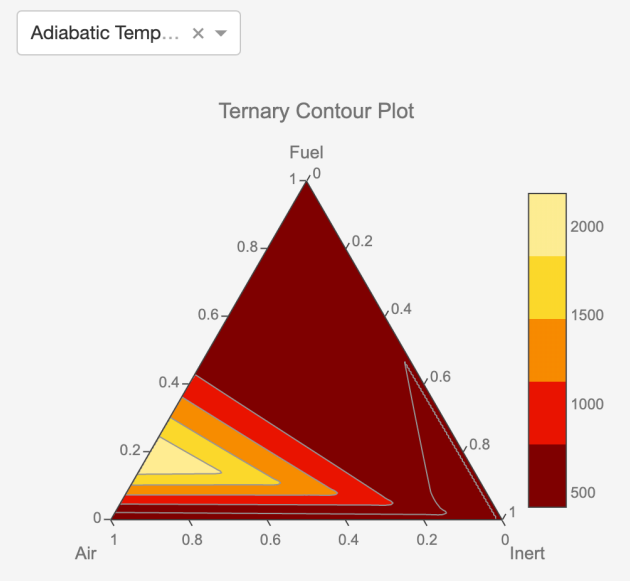


Figure 4.7: Adiabatic Temperature ternary contour plot

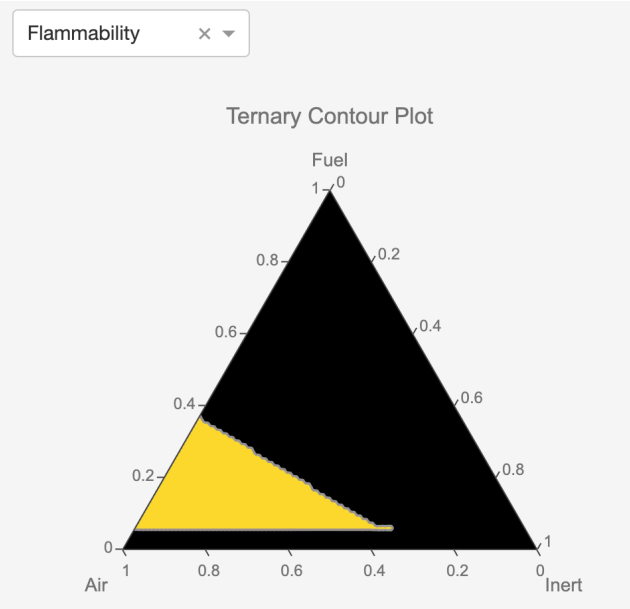


Figure 4.8: Flammability ternary contour plot

4.1.2.2 Flex Plot

Lastly, the dashboard also includes a *Flex Plot* that lets the user plot laminar flame speed, maximum adiabatic pressure, adiabatic temperature, or equivalence ratio vs either fuel ratio or equivalence ratio.

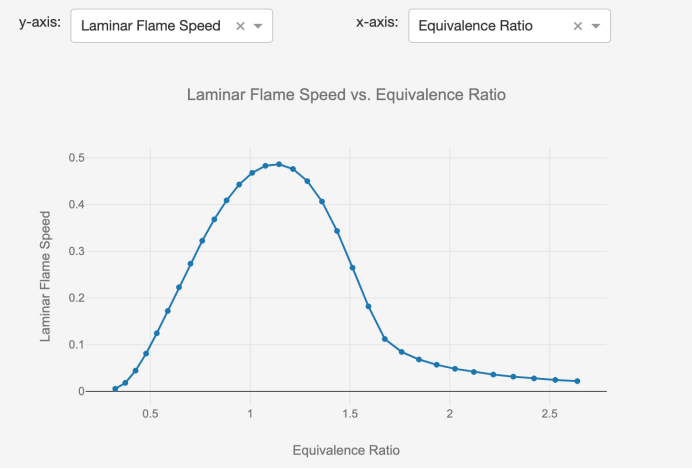


Figure 4.9: Summary table of LFL, UFL, flame speed and max pressure

Appendix

A0 – Summary of Cell Vent Gas Literature

Reference	Type of Cell	Cathode Chemistry	Electrolyte	Initial SOC (%)	Failure Environment	Failure Test
Kumai (1999) [12]	18650	LCO	PC:EMC: DEC:DMC	0%	Vacuum	Cycled/Overcharged/ Overdischarged at 25 C
Ohsaki (2005) [13]	Prismatic 633048	LCO	EC:EMC	OCH	N/P	Overcharged at 1 C Rate
Kong (2005) [14]	18650	LCO, LMO, LFP	EC:DEC	OCH	Ambient Conditions	Overcharged
Doughty (2005) [15]	18650	NCA	EC:PC:EMC & EC:EMC	100%	N/P	Thermal-Abuse (ARC)
Abraham (2006) [16]	18650	NCA	EC:EMC	100%	N/P	Thermal-Abuse (ARC)
Roth (2008) [17]	18650	NCA	EC:EMC	100%	Sealed Container	Thermal-Abuse (ARC)
Ribiere (2011) [18]	Pouch Cells	LMO	EC:DEC:DMC	0/50/100%	Air	Thermal-Abuse (Tewarson calorimeter)
Somandepalli (2014) [19]	Pouch Cells	LCO	ED:DEC	50/100/150%	Argon	Thermal Abuse (Combustion chamber)
Golubkov (2014) [20]	18650	LCO/NMC, NMC, LFP	DMC:EMC:EC & DMC:EMC:EC:PC	100%	Argon	Thermal Abuse (Reactor Chamber)
Larsson (2014) [21]	Battery Packs	LFP	N/P	0/50/100%	Air	Thermal Abuse (Fire Test Chamber)
Spinner (2015) [22]	18650	LCO	DMC:EC:PC	N/P	N/P	Thermal Abuse on Overcharged Cells(Fire Test Chamber)
Fu (2015) [23]	18650	LCO	N/P	0/50/65/ 70/100%	Air	Thermal-Abuse (Cone-calorimeter)
Yuan (2015) [24]	Prismatic	LFP	EC:DEC:EM	Varied from 100% to 190%	Air	Overcharged
Golubkov (2015) [25]	18650	NCA, LFP	EC:DMC:EMC:MPC & EC:DMC:EMC:PC	Varied from 0% to 143%	Inert Gas	Thermal Abuse
Sun (2016) [26]	18650	LCO, LMO, NMC, LFP	N/P	OCH	Air	Thermal Abuse on Overcharged Cells (Combustion Chamber)
Zheng (2016) [27]	Pouch Cells	LFP	EC:DMC:EMC	0%	N/P	Overdischarged (Glove Box)
FAA Study (2016) [28]	18650	LCO	N/P	0% thru 100%	Nitrogen 10 psia	Thermal-Abuse (Cone-calorimeter)
Lammer (2017) [29]	18650- (32A, 35E, MJ1)	NCA	N/P	100%	Inert Gas	Thermal Abuse
Larsson (2017) [30]	Pouch, Cylindrical	LCO, LFP, NCA-LATP	N/P	100%	Air	Thermal Abuse (Fire Test Chamber)
Fernandes (2018) [31]	Cylindrical, 26650	LFP	DMC:EMC: EC:PC	190%	Air Tight Polypropylene Enclosure	Overcharged (Explosion Proof Room)

A1 – Header Layout

```
html.Div(
            [
                html.Div(
                    [
                        html.Img(
                            src=("/assets/shield.png"),
                            style={"height": "60px", "width": "auto"}
                        )
                    ],
                    id="logo",
                    className='one-third column',
                ),
                html.Div(
                    [
                        html.H3(
                            'Building Deflagration',
                            style={"margin-bottom": "0px"}
                        ),
                        html.H5(
                            'Pressure-Time History',
                        )
                    ],
                    id="title",
                    className='one-half column',
                    style={"margin-bottom": "30px"}
                ),
                html.Div(
                    [
                        html.A(
                            html.Button("Hazard Analysis",
                                        id="hazard-analysis"),
                            href="/apps/hazard_analysis",
                            style={"float": "right"}
                        )
                    ],
                    className="one-third column",
                    id="button",
                ),
            ],
            id="header",
            className='row flex-display',
            style={"magin-bottom": "25px"}
        )
```

A2 – Dropdown Layout

```
main_dropdowns = html.Div(
    [
        html.P(
            'Pick a battery explosion experiment:',
            style={
                "font-size": "1.5em",
                "margin-left": "10px",
                "margin-bottom": "10px"}
        ),
        html.Div(
            id='db_data',
            children=get_main_data(),
            style={'display': 'none'}
        ),
        html.Div(
            [
                html.Label(
                    [
                        'Publication:',
                        dcc.Dropdown(
                            id='vent_ref_pub'
                        ),
                    ],
                    className="control_label"
                ),
                html.Label(
                    [
                        'Cell type:',
                        dcc.Dropdown(
                            id='vent_cell_types'
                        ),
                    ],
                    className="control_label"
                ),
                html.Label(
                    [
                        'Chemistry:',
                        dcc.Dropdown(
                            id='vent_cell_chemistry'
                        ),
                    ],
                    className="control_label"
                ),
```

```
                html.Label(
                    [
                        'Electrolyte:',
                        dcc.Dropdown(
                            id='vent_cell_electrolytes'
                        ),
                    ],
                    className="control_label"
                ),
                html.Label(
                    [
                        'SOC:',
                        dcc.Dropdown(
                            id='vent_cell_soc'
                        ),
                    ],
                    className="control_label"
                ),
                html.Div(
                    [
                        html.Button(id='clear_button',
                                    n_clicks=0,
                                    children='Clear',
                                    style={
                                        'position': 'relative',
                                        'top': '40%'}
                                    ),
                    ],
                ),
            ],
            className="pretty_container row",
            style={
                'width': '75%',
            }
        ),
    ]
)
```

A3 – Text Input Layout

```
html.Div([
                html.P('Room and vent dimensions:',
                       style={"font-size": "1.5em", "margin-left": "10px",
                              "margin-bottom": "10px"}
                ),
                html.Div(
                    [
                        html.Label(
                            [
                                'Spherical Room Radius:',
                                dcc.Input(
                                    id="vent_room_rad",
                                    type="number",
                                    min=0,
                                    placeholder="Radius (m)",
                                    className="control_label"
                                ),
                            ], className="control_label"
                        ),
                        html.Label(
                            [
                                'Vent Area:',
                                dcc.Input(
                                    id="vent_area",
                                    type="number",
                                    min=0,
                                    placeholder="Area (m^2)",
                                    className="control_label"
                                ),
                            ], className="control_label"
                        ),
                        html.Label(
                            [
                                'Vent Drag Coefficient (Cd):',
                                dcc.Input(
                                    id="vent_drag",
                                    type="number",
                                    min=0,
                                    placeholder="Drag Coeff",
                                    className="control_label"
                                ),
                            ], className="control_label"
                        ),
                    ], className="pretty_container row",
                       style={'width': '50%'}
                ),
        ]),
```

A4 – Graph Layout

```
html.Div(
            [
                html.Div(id='selected_experiment', style={'display':
'none'}),
                html.Div(id='gas_composition', style={'display': 'none'}),
                html.Div(
                    [
                        dcc.Graph(id='composition_plot')
                    ],
                    className='pretty_container four columns',
                ),
                html.Div(
                    [
                        dcc.Graph(id='explosion_plot')
                    ],
                    id="explosionGraphContainer",
                    className="pretty_container eight columns"
                )
            ],
            className="row"
        )
```

A5 – Database Helper Functions

```
import os
from pymongo import MongoClient
DB_URI = os.environ.get('DB_URI')
DB_NAME = os.environ.get('DB_NAME')

def find(collection, search={}, projection=None):
    """ Find items in a given collection matching a search.

    Parameters
    ----------
    collection : Unicode
        Name of te collection to search in.
    search : Dict
        The search dictionary with the search parameters.
    projection : Dict
        projection dictionary to specify or restrict fields to return.

    Returns
    -------
    List of items in the collection.
    """
    client = MongoClient(DB_URI)
    db = client[DB_NAME]
    return db[collection].find(search, projection)
```

A6 – Application Callbacks

```
@app.callback(
    [
        Output('vent_ref_pub', 'options'),
        Output('vent_cell_types', 'options'),
        Output('vent_cell_chemistry', 'options'),
        Output('vent_cell_electrolytes', 'options'),
        Output('vent_cell_soc', 'options')],
    [
        Input('db_data', 'children'),
        Input('vent_ref_pub', 'value'),
        Input('vent_cell_types', 'value'),
        Input('vent_cell_chemistry', 'value'),
        Input('vent_cell_electrolytes', 'value'),
        Input('vent_cell_soc', 'value')
    ])
def update_dropdowns(data, publication, cell_type, chemistry, electrolyte,
                     soc):
    """ Update gas dropdown databa based on selections. """
    df = pd.DataFrame(json.loads(data))

    if publication:
        df = df[df['Publication'] == publication]
    if cell_type:
        df = df[df['Format'] == cell_type]
    if chemistry:
        df = df[df['Chemistry'] == chemistry]
    if electrolyte:
        df = df[df['Electrolyte'] == electrolyte]
    if soc:
        df = df[df['SOC'] == soc]

    fields = ['Publication', 'Format', 'Chemistry', 'Electrolyte', 'SOC']
    results = []
    for field in fields:
        result = list(df[field].unique())
        results.append(make_options(result))
    return results
```

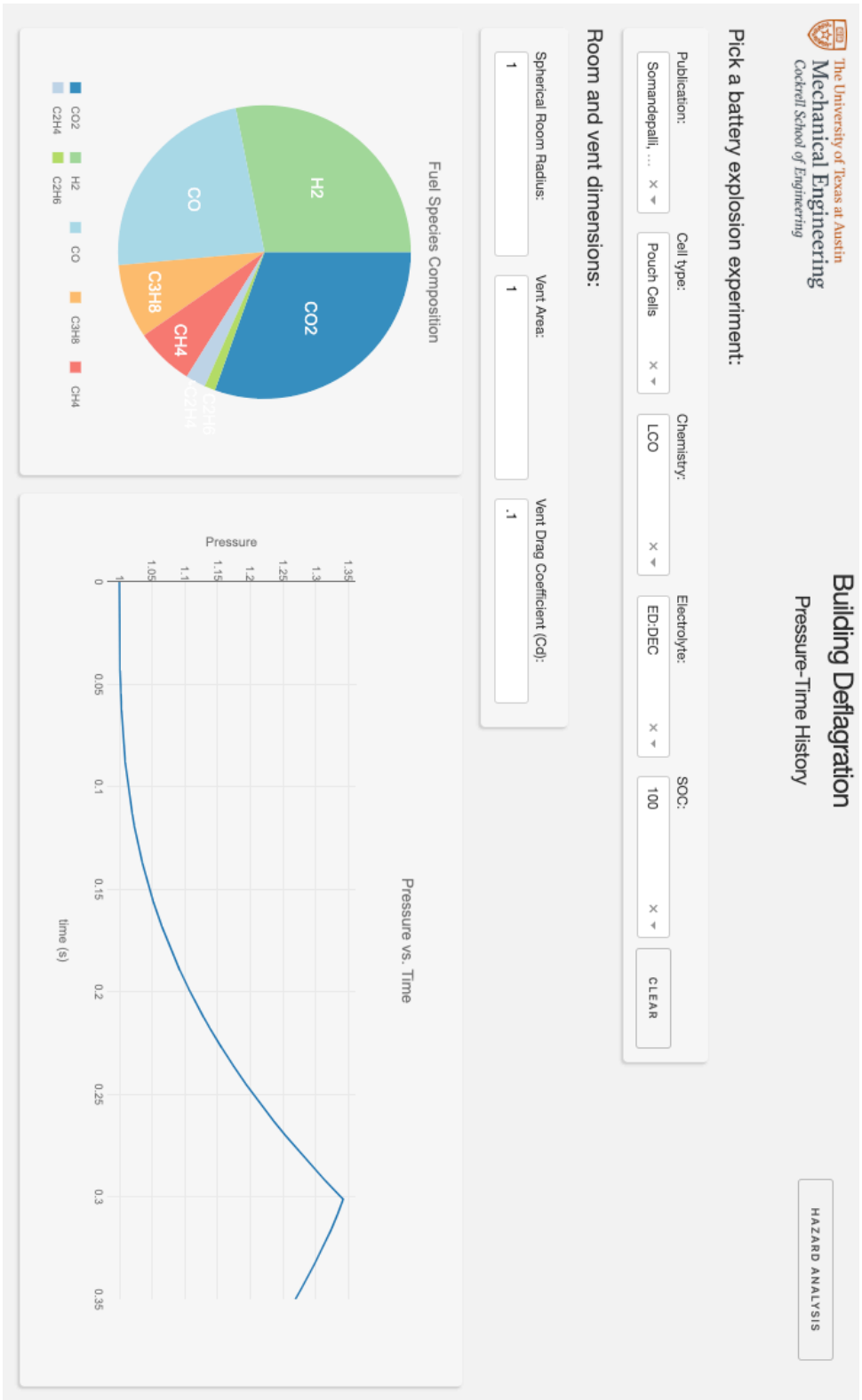
```
@app.callback(
    [
        Output('vent_ref_pub', 'value'),
        Output('vent_cell_types', 'value'),
        Output('vent_cell_chemistry', 'value'),
        Output('vent_cell_electrolytes', 'value'),
        Output('vent_cell_soc', 'value')],
    [
        Input('clear_button', 'n_clicks')],
    [
        State('vent_ref_pub', 'value'),
        State('vent_cell_types', 'value'),
        State('vent_cell_chemistry', 'value'),
        State('vent_cell_electrolytes', 'value'),
        State('vent_cell_soc', 'value')]
)
def clear_dropdowns(n_clicks, publication, cell_type, chemistry, electrolyte,
                    soc):
    """ Clear dropdown menus. """
    if n_clicks > 0:
        return [], [], [], [], []
    else:
        return publication, cell_type, chemistry, electrolyte, soc

@app.callback(
    Output('selected_experiment', 'children'),
    [
        Input('vent_ref_pub', 'value'),
        Input('vent_cell_types', 'value'),
        Input('vent_cell_chemistry', 'value'),
        Input('vent_cell_electrolytes', 'value'),
        Input('vent_cell_soc', 'value')
    ],
    [State('selected_experiment', 'children')])
def update_selected_experiment(publication, cell_type, chemistry,
                               electrolyte, soc, current_experiment):
    """ Update search experiment data based dropdown selections. """
    if all([publication, cell_type, chemistry, electrolyte, soc]):
        dct = {'Publication': publication, 'Format': cell_type,
               'Chemistry': chemistry, 'Electrolyte': electrolyte, 'SOC':
soc}
        search = json.dumps(dct)
    elif any([publication, cell_type, chemistry, electrolyte, soc]):
        search = current_experiment
    else:
        search = None

    return search
```

```python
@app.callback(
    Output('gas_composition', 'children'),
    [Input('selected_experiment', 'children')])
def update_gases(selected_experiment):
    """ Update gas data based on dropdown selections. """
    if selected_experiment:
        search = json.loads(selected_experiment)
        _clean_search_dict(search)
        search = _add_search_filter(search)
        values = get_unique(MAIN_COLLECTION, field='Gases', search=search)
        gases = values[-1] if values else ''
        return json.dumps(gases)
```

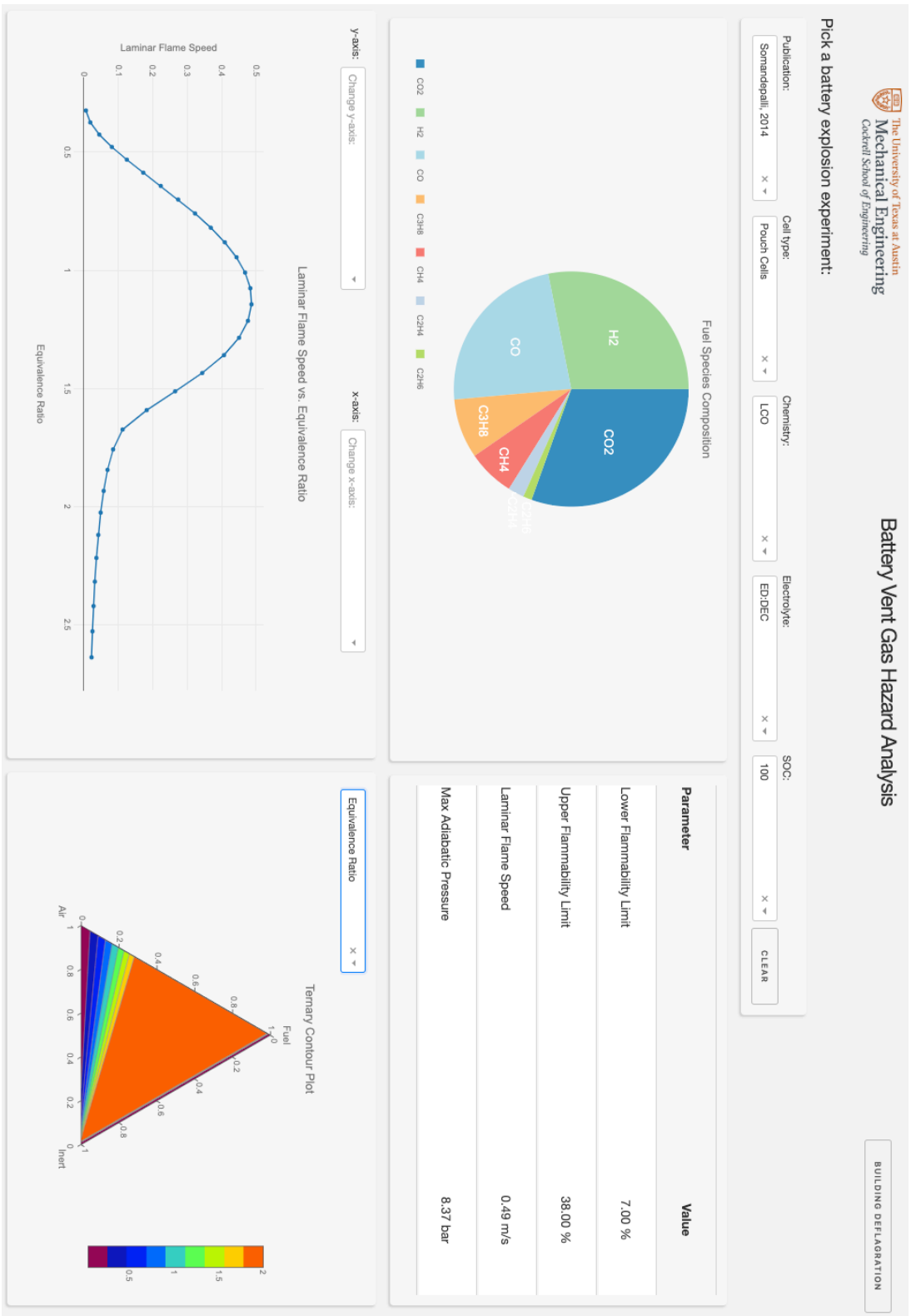
A7 – Dockerfile

```
FROM conda/miniconda3

# Grab environment.yml
ADD ./environment.yml /tmp/environment.yml

# Add our code
ADD ./firedash /opt/firedash/
WORKDIR /opt/firedash

RUN conda update -q conda

# Install conda dependencies
RUN conda env create -n fire -f /tmp/environment.yml

CMD /usr/local/envs/fire/bin/gunicorn --bind 0.0.0.0:$PORT index:app.server
```

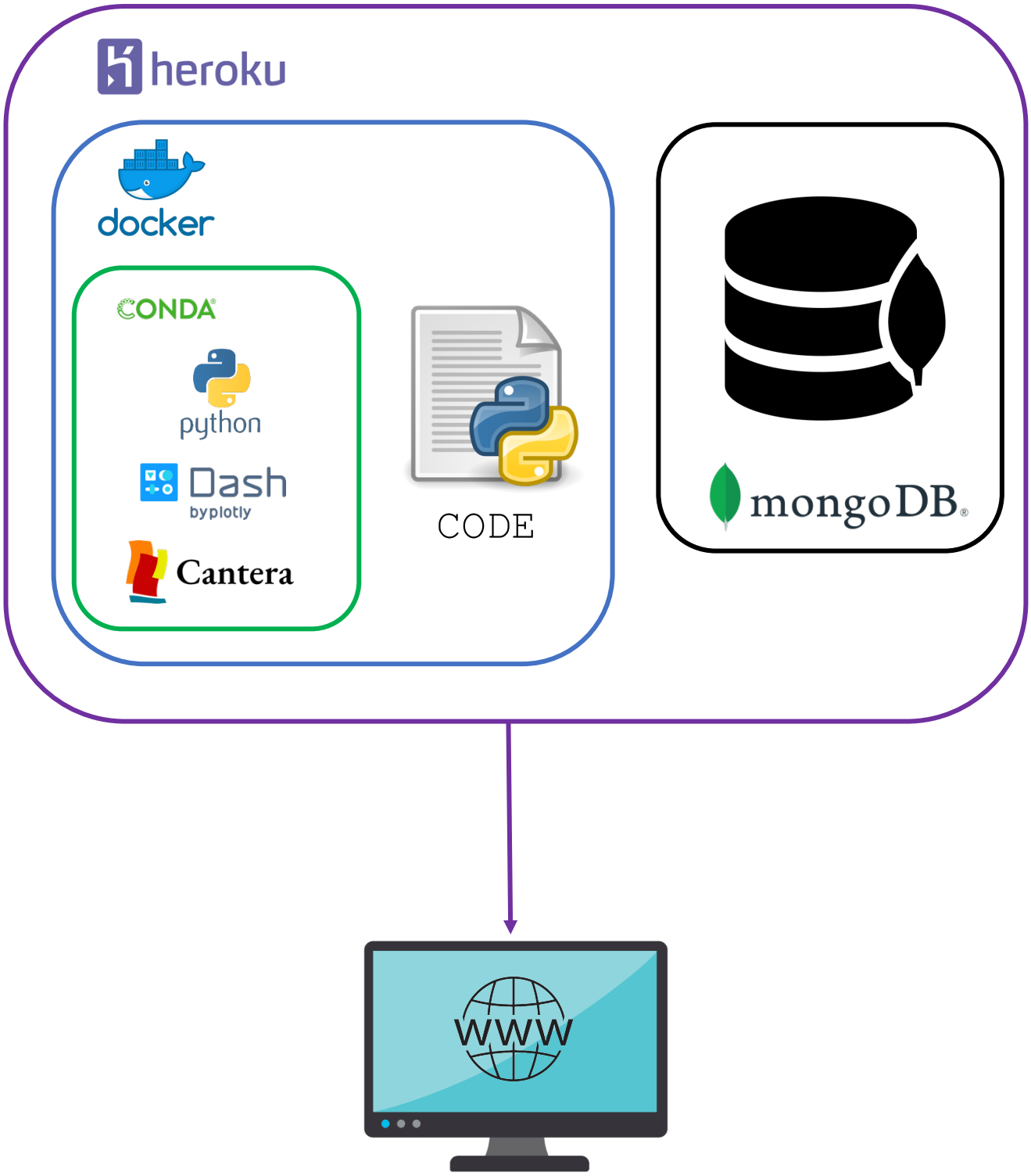
A8 – Building Deflagration Dashboard



A9 – Battery Vent Gas Hazard Analysis Dashboard



A10 – Application Diagram



Glossary

Acronyms

AWS – Amazon Web Services

GCP – Google Cloud Platform

API – Application Programming Interface

PaaS – Platform as a Service

UI – User Interface

html – dash_html_components

dcc – dash_core_components

CRUD – Create, Read, Update, and Delete operations

CLI – Command Line Interface

OS – Operating System

LFL – Lower Flammability Limit

UFL – Upper Flammability Limit

References

Baird, Austin R., Erik J. Archibald, Kevin C. Marr, and Ofodike A. Ezekoye. “Explosion Hazards from Lithium-Ion Battery Vent Gas.” Journal of Power Sources 446 (January 15, 2020). https://doi.org/10.1016/j.jpowsour.2019.227257.

Brown, K. (2019, November 13). What Is GitHub, and What Is It Used For? Retrieved from https://www.howtogeek.com/180167/htg-explains-what-is-github-and-what-do-geeks-use-it-for/.

Conda. (2019, November 20). Retrieved from https://docs.conda.io/en/latest/.

Deploying Django to Heroku With Docker. (2019, October 15). Retrieved from https://testdriven.io/blog/deploying-django-to-heroku-with-docker/.

Dockerfile reference. (2019, November 28). Retrieved from https://docs.docker.com/engine/reference/builder/.

Getting Started on Heroku with Python. (2019, October 15). Retrieved from https://devcenter.heroku.com/articles/getting-started-with-python.

Heroku. (2019, October 15). Retrieved from https://www.heroku.com/.

(2019, October 1). Retrieved from https://dash.plot.ly/introduction.

Part 2. Layout. (2019, October 1). Retrieved from https://dash.plot.ly/getting-started.

Wikimedia Foundation. (2019, November 20). Platform as a service. Retrieved from https://en.wikipedia.org/wiki/Platform_as_a_service.

Scientific Applications - The Hitchhiker's Guide to Python. (2019, November 20). Retrieved from https://docs.python-guide.org/scenarios/scientific/.

The Procfile: Heroku Dev Center. (2019, October 20). Retrieved from https://devcenter.heroku.com/articles/procfile.

The Python Tutorial. (2019, November 20). Retrieved from https://docs.python.org/3/tutorial/.

What Is MongoDB? (2019, October 15). Retrieved from https://www.mongodb.com/what-is-mongodb.

Vita

Juan Trejo was born in Mexico City, Mexico to Elsa Marcela Martinez and Juan Trejo and was raised in Puebla, Mexico. He graduated from Instituto Oriente de Puebla High School in May 2008 and received his Bachelor of Science in Mechanical Engineering from The University of Texas at Austin in December of 2012. Upon receiving his undergraduate degree, he began his career working at Cummins as a Product Validation Engineer in Columbus, Indiana. After three years in this role, he returned to the University of Texas at Austin to receive his Masters of Science in Mechanical Engineering. He currently works as a Scientific Software Developer at Enthought, and enjoys exploring Austin with his wife, Tori, and dog, Mylo on the weekends. Upon graduation, he will continue working at Enthought as a Scientific Software Developer.

Email address: jtrejo13@utexas.edu

This dissertation was typed by Juan Trejo Martinez.